

Formal Verification of Control Systems' Properties with Theorem Proving

Dejanira Araiza-Illan and Kerstin Eder

Dept. Computer Science

University of Bristol

Bristol, UK

{Dejanira.Araizaillan, Kerstin.Eder}@bristol.ac.uk

Arthur Richards

Dept. Aerospace Engineering

University of Bristol

Bristol, UK

Arthur.Richards@bristol.ac.uk

Abstract

This paper presents the deductive formal verification of high-level properties of control systems with theorem proving, using the Why3 tool. Properties that can be verified with this approach include stability, feedback gain, and robustness, among others. For the systems, modelled in Simulink, we propose three main steps to achieve the verification: specifying the properties of interest over the signals within the model using Simulink blocks, an automatic translation of the model into Why3, and the automatic verification of the properties with theorem provers in Why3. We present a methodology to specify the properties in the model and a library of relevant assertion blocks (logic expressions), currently in development. The functionality of the blocks in the Simulink models are automatically translated to Why3 as 'theories' and verification goals by our tool implemented in MATLAB. A library of theories in Why3 corresponding to each supported block has been developed to facilitate the process of translation. The goals are automatically verified in Why3 with relevant theorem provers. A simple first-order discrete system is used to exemplify the specification of the Simulink model, the translation process from Simulink to the Why3 formal logic language, and the verification of Lyapunov stability.

1 Introduction

The use of formal methods allows the production of reliable and certified trustworthy autonomous systems [12, 25] in a more automatic manner, facilitating the processes of verification and validation, and introducing methodologies for systems design towards verification. Verification has become essential for safety-critical autonomous systems. Autonomous systems have different compositional levels [12]: agent (high-level planning and decision making), control (neural networks, controllers or control systems) and hardware implementation. Many proposed modelling formalisms and formal verification examples deal with the first layer only [9, 26, 30], or the first and second combined in a high-level abstraction (e.g., hybrid systems). For the latter, some approaches use theorem proving [19]. Other approaches translate the systems to decidable automata models and apply model checking [10, 18, 20], but these are not easily applicable to real-valued operations due to problems with scalability. Additionally, optimisation theory has been used to verify functionality and robustness of controllers for autonomous systems [14, 29].

The design of control systems typically begins with formal analysis followed by numerical implementation in a simulation tool like Simulink. Numerical simulations then test for correct behaviour before the implementation is deployed. In some cases, automatic code generation [24] is used to derive code directly from the simulation model. In this paper, we propose the application of deductive formal verification methods on the implementation of the controller in Simulink. This allows greater confidence in the correctness of the model with respect to its requirements. In particular, the paper provides a way to translate Simulink models into the Why3 logic language to automatically prove results such as decrease

of a Lyapunov function. One of the future aims of proposed approach is to verify the correctness of controllers based on an optimiser; for example, predictive controllers for UAV guidance [21, 22].

Why3 is a free and open source tool that interfaces with different theorem provers, particularly their Satisfiability Modulo Theory (SMT) solvers. SMT solvers are extensions of SATisfiability (SAT) solvers that can accommodate real numbers, integers, and other domains (theories), and are thus well-suited to prove control systems properties [8]. Automatic theorem provers currently supported by Why3 include Alt-Ergo, CVC3, CVC4, E-prover, Gappa, Simplify, SPASS, Vampire, veriT, Yices and Z3, along with the interactive provers Coq and PVS [11]. An SMT solver automatically computes the satisfiability of a logic formula based on a range of provided axioms and definitions.

The closest prior work to this paper in implementation is Simcheck [23], which uses SMT solvers to perform type checking on a Simulink model with custom annotations. Stability based on Nichols plots has been verified for a system modelled in Simulink using the MetiTarski theorem prover [8]. Other approaches generate annotated C code from the model, then apply theorem proving to verify stability properties [15, 17]. Functional correctness of auto-generated code from Simulink models has also been formally verified through theorem provers [1, 2, 6, 28] and model checking [27].

The main novel contributions in this paper are:

- A methodology to specify high-level properties for control systems in Simulink, supplemented by our custom blocks.
- An automatic translation tool from Simulink to Why3, for formal verification of the properties.

Our aim in pursuing this approach is to ease the inclusion of verification at design time, by incorporating it at the block diagram level where system interconnections and insight are best expressed.

In our proposed approach, higher-order logic requirements are incorporated and verified as Simulink signals [5]. Note that this enables the same requirements to be tested by numerical simulation. This approach is partly inspired by the Open Verification Library (OVL) [16], a library of modules that act as property checkers and can be placed into a hardware design where they are connected to regular modules using the circuit signals. For example, our new custom ‘Require’ block, containing the built-in ‘Assert’ block, incorporates pre- and post-conditions into the Simulink model. Then the automatic translation process to Why3 identifies the ‘Require’ blocks as goals for theorem proving. Meanwhile, each other block in the model is translated into the axioms it asserts relating its input and output signals. Finally, Why3 is invoked to run a chosen SMT solver on the translated ‘theory’ and prove the goals, or otherwise.

The rest of the paper presents relevant aspects of the Why3 language, and our proposed approach. Section 2 introduces our developed assertion blocks, particularly the ‘Require’ block. Section 3 presents a review of the Why3 logic language components. Section 4 describes the translation process from Simulink to Why3, based on our developed theories in Why3 for different blocks. The translation is exemplified by a discrete system. Section 5 explains the verification of Lyapunov stability for the same discrete system. The presented approach is discussed in the same section. Finally, the conclusion is presented in Section 6.

2 Assertion Blocks

We are developing a set of assertion blocks analogous to OVL for hardware verification [16], to add to Simulink models to test and verify high-level requirements through specifications (as assertions) over the signals in the models. The OVL was developed to facilitate the addition of verification conditions (predefined logic expressions that can act as assertions, e.g. ‘event X always happens’, assumptions or coverage points) to any hardware design, for assertion-based verification [13] or formal verification. The OVL checkers, written in Hardware Description Languages, receive names of the signals of interest as inputs, and are being monitored at simulation time when assertion violations are being recorded. The same assertions can serve as goals for formal verification.

The new blocks represent a range of assertions (logic expressions) to describe properties of interest related to control systems over signals, and others refer to structure for the assertions. Available Simulink assertion and logic blocks like ‘Compare To Zero’, ‘Compare To Constant’ and ‘Assertion’ are also used as part or in conjunction with the new assertion blocks. The ‘Require’ block denotes a requirement in a Hoare triple form [3]

$$\{\text{preconditions}\} \rightarrow [\text{model}]\{\text{postconditions}\}, \quad (1)$$

where the logic expressions for the preconditions and postconditions are translated from the assertion blocks connected to the ‘Require’ block. The translator automatically produces the verification goals

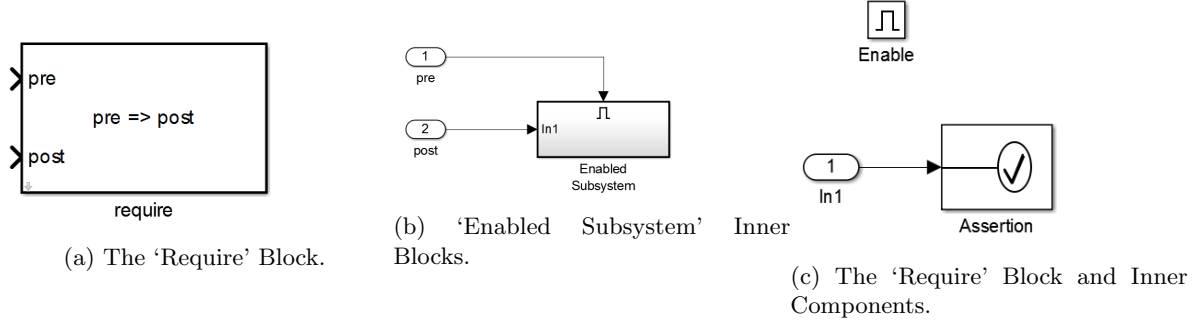


Figure 1: A First-Order System Specified with Assertion Blocks in Simulink.

for the theorem prover from these blocks. The 'Require' block is an enabled subsystem with an 'Assert' block inside (Fig. ??). They use native Simulink and MATLAB formats, thus allowing numerical testing in simulation and coverage assessment.

In the first-order system of Fig. 1, we have implemented a Lyapunov function $V(x) = x^2$ for the analysis of stability. Three 'Require' blocks ('require', 'require1' and 'require2') describe stability properties parametrised by signals in the model:

- 'require': $x \neq 0 \rightarrow V(x(k)) - V(x(k-1)) < 0$. If the signal x is different from 0, the signal leaving the 'Difference' block should be negative.
- 'require1': $x \neq 0 \rightarrow V(x) > 0$. If the signal x is different from 0, then the signal $V(x)$ should be positive.
- 'require2': $x = 0 \rightarrow V(x) = 0$. If the signal x is 0, then the signal $V(x)$ should be zero.

More complex assertions are built-up from multiple basic blocks, which guarantees their translatability into Why3 (using the predefined theories for basic blocks) and preserves the modularity in Simulink. Properties can be verified in independent sections of blocks, or even within blocks inside subsystems. Individual proofs can be assembled into a hierarchical proof structure, to provide conclusions at a higher level from the sequence and relations of the lower-level proofs.

3 Review of Why3 Syntax

The Why3 logic language is based on polymorphic types and first-order logic, although higher-order logic syntax is allowed [4, 11]. The WhyML language targets the formal verification of programs, by including loops and other control structures. The Why3 logic language is sufficient to express and verify block diagrams from Simulink due to their declarative nature. The translation from Simulink to Why3 logic expressions is based on the following Why3 components:

- Theories: blocks of logic expressions with types, functions, predicates, axioms, lemmas and verification goals that can be used or cloned in a modular manner. More complex theories can be formed from basic theories, inheriting the axioms, lemmas, functions and declarations. They contain the components for a proof.

```

theory Product_int
  use import int.Int
  use import real.RealInfix

  function in1 int: real
  function in2 int: real
  function out1 int: real

  axiom v: forall k:int. out1 k = in1 k *. in2 k
  axiom c1: forall k:int. in1 k >. 0.0 /\ in2 k >. 0.0 -> out1 k >. 0.0
  axiom c2: forall k:int. in1 k <. 0.0 /\ in2 k <. 0.0 -> out1 k >. 0.0
end

```

Figure 2: Theory for the ‘Product’ Block.

```

theory <Name>
...
end

```

- Functions: operations over data, that can be recursive and accept different types (e.g., integers, real, Boolean). The parameters are expressed in curried syntax, where a function $f(x)$ becomes $f\ x$.

```
function <name> <inputs> <types> : <outputs> <types>
```

- Axioms and lemmas: logic expressions to help a proof.

```
axiom/lemma <name>: <logic expression>
```

- Goals: logic expressions to prove.

```
goal <Name>: <logic expression>
```

- Theories can be used (same symbols) and cloned (no reuse of symbols) in other theories. Some basic theories are available, including real numbers, integers, floating-point and lists.

```
use import <file_name>.<Name>
```

```
clone <file>.<Name> as <New_name> with function
<name> = <new_name>
```

We have developed a library of theories that provide axioms applicable to the input-output behaviour corresponding to popular Simulink blocks, including arithmetic and logic, to facilitate the translation into Why3. The theories are cloned following the components of a block diagram, and the axioms within each theory are parametrised according to signal names and specific values (constants, input ranges or initial conditions). The theory developed for the ‘Product’ block is shown in Fig. 2. In the future, additional linear algebra and control systems related theories will be developed as needed in order to verify more complex systems and properties, as in [15].

In the `Product_block` theory, the theories of the integer and real numbers are imported first. Then, two functions are used for the inputs and another for the output. They receive an integer (time sample) and produce a real (the signal value at that time). Three axioms describe the multiplication and some sign properties: for all sampled times, `forall k:int.`, the output signal is equal to the multiplication of the inputs (`axiom v`); the multiplication of two positive numbers produces a positive number (`axiom c1`); and the multiplication of two negative numbers produces a positive number (`axiom c2`). The algebraic and logic symbols are used with a dot when a theory mixes integer and real numbers (i.e., both theories are imported in the same theory), whereas if a theory only uses integer or real numbers, the symbols are used without the dot [4].

```

theory CompareToZero_neq_int
  use import int.Int
  use import real.RealInfix
  use import bool.Bool

  function in1 int: real
  function out1 int: bool

  axiom v1: forall k:int. out1 k = True -> in1 k <>. 0.0
  axiom v2: forall k:int. out1 k = False -> in1 k = 0.0
end

```

Figure 3: Theory for the ‘Compare To Zero’ Block, for the $\sim = 0$ Case.

The theory developed for the ‘Compare To Zero’ block with condition case $\sim = 0$ (not equal to zero) is shown in Fig. 3.

In the `CompareToZero_block_ne0` theory, the theories of the integer, real and Boolean numbers are imported first. Then, a function is defined for the input and another for the output. The input receives an integer (time sample) and produces a real value (the signal value at that time), whereas the output produces a Boolean value (`True` or `False`). Two axioms describe the conversion from a real valued signal (input) to a Boolean (output), according to the comparison criterion: if the input value is different to zero (`axiom v1`), the output is `True`; and if the input value is zero the output is `False`.

4 Translation from Simulink to Why3

The automatic translation process from Simulink to Why3 converts the signals and blocks into higher-order logic predicates [5]. For example, the functionality of a ‘Product’ block over two inputs and an output signal in discrete intervals of time (k) is expressed in higher-order logic [5] as

$$Product(in_1, in_2, out_1) \equiv \forall k. out_1(k) = in_1(k) * in_2(k). \quad (2)$$

The proposed and implemented procedure for the translation of systems as block diagrams in Simulink into Why3 language consists of the following steps, executed automatically in MATLAB when calling the translator function:

1. Identification of the specification blocks from the rest of the model. For example, ‘Require’ blocks and their associated preconditions and postconditions, where the latter two are directly connected to the respective labelled inputs of the ‘Require’ blocks.
2. The Abstract Syntax Tree (AST) [7,23] of the model is computed automatically. We implemented a program that identifies the predecessors and successors of each block, using the block manipulation functions available in MATLAB. Each AST is translated into a theory in Why3, named after the model to be verified. Relevant numerical theories including real numbers, integers and Boolean expressions are imported.
3. Then, every signal that leaves a block is identified, named and translated to real or Boolean functions. Functions that receive an integer represent discrete signals (i.e., signals of the form $y(k)$ where k is the time step):

```
function <block_name>_op<n> int: <real/bool>
```

where $\langle n \rangle$ is the corresponding output port (in case the block has more than one), `int` indicates if the signal is discrete, and `real/bool` indicates if the signal is numeric or Boolean, respectively. In this paper, all the signals in Simulink are scalar and discrete, although the translation and verification will be extended to continuous time in the future.

4. Afterwards, the theories corresponding to all the blocks in the system are cloned automatically, with parametrisation (e.g., gains, constants) as added axioms. The cloning of some structural specification blocks like ‘Require’ is excluded, as the main purpose of these blocks in the translation is to give structure to the verification goals (e.g., Hoare triple inference form).

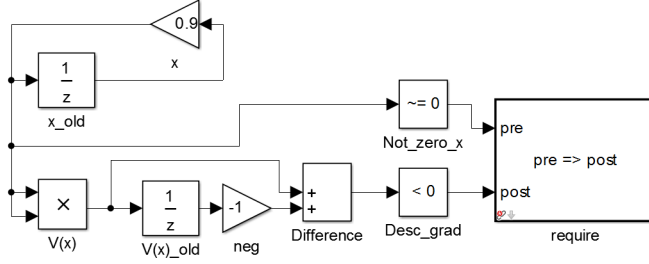


Figure 4: Reduced First-Order System in Simulink.

```

theory M_firstorder
  use import int.Int
  use import real.RealInfix
  use import bool.Bool

  function difference_op1 int: real
  function vx_op1 int: real
  function vx_old_op1 int: real
  function neg_op1 int: real
  function x_op1 int: real
  function x_old_op1 int: real
  function not_zero_x_op int: bool
  function desc_grad_op int: bool

  clone simulink.Sum_int as Difference with function in1 = vx_op1, function in2 = neg_op1, function out1 = difference_op1

  clone simulink.Product_int as Vx with function in1 = x_op1, function in2 = x_op1, function out1 = vx_op1

  clone simulink.UnitDelay_int as Vx_old with function in1 = vx_op1, function out1 = vx_old_op1

  clone simulink.Gain_int as Neg with function in1 = vx_old_op1, function out1 = neg_op1
  axiom neg_gain: Neg.gain = -.1.000000

  clone simulink.Gain_int as X with function in1 = x_old_op1, function out1 = x_op1
  axiom x_gain: X.gain = 0.900000

  clone simulink.UnitDelay_int as X_old with function in1 = x_op1, function out1 = x_old_op1

  clone simulink.CompareToZero_neq_int as Not_zero_x with function in1 = x_op1, function out1 = not_zero_x_op

  clone simulink.CompareToZero_l_int as Desc_grad with function in1 = difference_op1, function out1 = desc_grad_op

  goal G1 : forall k: int. not_zero_x_op k = True -> desc_grad_op k = True
end

```

Figure 5: Theory for the First Order System in Fig. 4.

- Finally, the verification goals are added following structures like the Hoare triple form from the ‘Require’ blocks. The preconditions and postconditions are added as desired Boolean signal outcomes in the Hoare triple:

```
goal <Name>: <time>. <precond.> -> <postcond.>
```

If the Simulink model has nested generic subsystems (i.e., grouping blocks with the purpose of providing modularity), each subsystem is translated into a theory from the computation of its internal AST and verification goals, using the previously described procedure. These theories are then cloned into other theories, following the flow diagrams of upper levels.

Consider the first-order discrete system in Simulink shown in Fig. 4, a simplification of Fig. 1. The translator produces the theory in Fig. 5 automatically.

The name of the theory, in the first line, corresponds to the name of the model file in Simulink. Then, the theories of the integer, real and Boolean numbers are imported. Functions corresponding to the signals that come out of the blocks are added to the theory, specifying if they correspond to real or Boolean signals, and following the convention for the naming described before.

Subsequently, the theories of the different blocks in the Simulink model are cloned and parametrised from our library (contained in a file named ‘simulink’) mentioned in Section 3. The order of the blocks is given by the Simulink automatic identification numbering: the ‘Sum’ block, the ‘Product’ block (the Lyapunov function), the ‘Unit Delay’ block of the Lyapunov function, the ‘Gain’ block to achieve the

subtraction of the values of the Lyapunov function combined with the ‘Sum’ block, the ‘Gain’ block and the ‘Unit Delay’ of the system. When a theory from the library is cloned, its name is changed to the name of the block. The input and output functions of the new theory are renamed according to the signals connected to the respective block. Examples of value parametrisation in the cloning are the axioms corresponding to the gains `Neg_gain` and `x_gain`, for the ‘Gain’ blocks in the model.

The theories corresponding to the comparison blocks (the two ‘Compare To Zero’ blocks) are cloned considering the parameters assigned in the model, i.e., calling the respective theory that corresponds to a comparison of any of the possible types $\{==, \sim, >, <, >=, <= \}$. Finally, the verification goal is added by connecting the precondition and postcondition following the structure of the Hoare triple, `not_zero_x_op1 k = True -> desc_grad_op1 k = True`.

5 Verification of a First-Order System

Consider the same first-order linear discrete system as in Fig. 4, for a single dimension (scalar),

$$x(k+1) = ax(k), \quad (3)$$

and a metric for stability given by a quadratic Lyapunov function, that is always positive regardless of its input,

$$V(x) = x^2. \quad (4)$$

A system like the one in our example is considered stable if its trajectory in time converges to a region or point in the state space, starting from different initial conditions. A way to prove stability of the system in Fig. 4 is the existence of a Lyapunov function that preserves its expected properties of positivity, and that will decrease (if the system is stable), stay in the same value (if the system is marginally stable), or increase (if the system is unstable). Stability is proved in our examples for different values of the gain a computing the discrete gradient descent of the Lyapunov function (the difference between the value of the function in the current time interval and in the previous interval).

The conditions to prove stability through the signals has been expressed in two assertions added as blocks to the Simulink model: when the signal x is not zero (precondition), if the system is stable the gradient of the Lyapunov function should be negative (postcondition),

$$x(k) \neq 0 \rightarrow V(x(k)) - V(x(k-1)) < 0 \quad (5)$$

as shown in Fig. 4. Thus, a failure of proof is expected for the verification of the goal `goal G1` when the gain is ≥ 1 or ≤ -1 , as the implication would lead to $\text{True} \rightarrow \text{False}$, which is in turn False . The validity of the proof is expected when the gain is < 1 , > -1 or $= 0$, as the first two lead to $\text{True} \rightarrow \text{True}$, and the later to $\text{False} \rightarrow \text{False}$, all implications resulting in True . These conclusions are derived from the truth table for the logical implication operation.

5.1 Assertion Checks in Simulation

The stability assertions specified in the Simulink model have been checked in simulation, for different gain values in the system loop. If the system does not comply with the specified property, the ‘Assert’ block inside the ‘Require’ block will fire up a flag when running the Simulink model. The results for different gains are shown in Table 1, conforming to our predictions: failure when the gain is ≥ 1 and ≤ -1 (thus the system is unstable), and success in the checks when the gain is < 1 , > -1 (thus the system is stable), highlighted in the table. When the gain is $= 0$, the ‘Assert’ block inside the ‘Require’ is not active and the check does not take place, due to the disabling action of the precondition in the ‘Enabled Subsystem’, as the value of the signal $x(k)$ is 0 and thus the precondition is false.

5.2 Proofs in Why3

The goal has been verified for different gains using the CVC3 SMT solver, compatible for theories with real and integer numbers, linear arithmetic, equalities and inequalities. The results of the verification are shown in Table 2. The highlighted results indicate proved stability (not including marginal), from the validity of both the precondition and postcondition, and the ‘Unknown’ results show that stability could not be proved for those gains. These results match the assertion checks presented in Table 1. CVC3 automatically computed the validity of the verification goals in less than 150 seconds, our specified time limit in Why3.

Table 1: Checks with Different Gains

| GAIN | CHECK RESULT |
|---------------|--------------|
| -1.1 | Fail |
| -1.0 | Fail |
| -0.5 | Pass |
| 0.0 | Not checked |
| 0.8 | Pass |
| 0.9 | Pass |
| 0.9999 | Pass |
| 1.0 | Fail |
| 1.1 | Fail |
| 2.0 | Fail |

Table 2: Verification with Different Gains

| GAIN (<code>x.gain</code>) | G1: stability $x(k) \neq 0 \rightarrow$ $V(x(k)) - V(x(k-1)) < 0$ | Prec. | Postc. |
|---------------------------------|---|-------------|-------------|
| -1.1 | Unknown | True | False |
| -1.0 | Unknown | True | False |
| -0.5 | Valid | True | True |
| 0.0 | Valid | False | False |
| 0.8 | Valid | True | True |
| 0.9 | Valid | True | True |
| 0.9999 | Valid | True | True |
| 1.0 | Unknown | True | False |
| 1.1 | Unknown | True | False |
| 2.0 | Unknown | True | False |

The case study presented in this paper showcases our approach to formally prove properties of interest, from the translation of Simulink into Why3. The translation is performed automatically for a subset of Simulink blocks. This subset is currently in expansion. Different theorem provers can be selected in Why3 for the proof, according to their capabilities. The main disadvantage of Why3 is the lack of feedback about the failure in the proof, compared to the production of counterexamples in model checking tools. Nevertheless, the tool chain can be extended to incorporate equivalent mechanisms for feedback.

6 Conclusion

This paper presented an approach to formally verify high level properties of interest of control systems (e.g., stability) as Simulink models in a more automatic manner in Why3 (using theorem proving tools). The Simulink model is specified or annotated with assertions in blocks from our proposed reference library. The Simulink models are translated automatically into the Why3 logic syntax, from the signals in time and the block connections (computing an abstract syntax tree). The translation process is helped by our library of theories in Why3 that correspond to the functionality of a set of Simulink blocks. The assertion blocks are translated into verification goals, and then proved by calling different theorem provers in Why3 according to their capabilities (theories).

An example of the translation and verification was provided in the form of a first-order discrete

system, coupled with a Lyapunov function as a metric for stability. The verification goals were proved or otherwise, according to the expected stability results for different parameters (gain and initial values), using the CVC3 theorem prover (or its embedded SMT solver). Assertion checks were performed over the Simulink model in simulation, to show the useful duality of our proposed specification approach.

Immediate future work includes extending our approach to multidimensional problems by incorporating linear algebra theories in Why3. Other future work includes the expansion of our libraries of assertion blocks and theories for supported Simulink blocks, for other high-level properties of interest (e.g., robustness) and stability methods. We are intending to apply our methodology and tools to a diverse range of systems.

Acknowledgment

The work presented in this paper was supported by the EPSRC grant EP/J01205X/1 RIVERAS: Robust Integrated Verification of Autonomous Systems.

References

- [1] A. Anta, R. Majumdar, I. Saha, and P. Tabuada. Automatic verification of control system implementations. In *Proc. EMSOFT*, pages 9–18, Scottsdale, AZ, USA, October 2010.
- [2] R. Arthan, P. Caseley, C. O’Halloran, and A. Smith. ClawZ: control laws in Z. In *Proc. ICFEM*, pages 169–176, York, UK, September 2000.
- [3] R. Arthan, U. Martin, and P. Oliva. A Hoare logic for linear systems. *Formal Aspects of Computing*, 25:345–363, 2011.
- [4] F. Bobot, J.C. Filliâtre, C. Marché, G. Melquiond, and A. Paskevich. *The Why3 Platform*. University Paris-Sud, CNRS, Inria, March 2013.
- [5] A. Camilleri, M. Gordon, and T. Melham. Hardware verification using higher-order logic. Technical Report 91, University of Cambridge, Computer Laboratory, September 1986.
- [6] A. Cavalcanti and P. Clayton. Verification of control systems using Circus. In *Proc. ICECCS*, Stanford, CA, USA, 2006.
- [7] C. Chen, J.S. Dong, and J. Sun. A formal framework for modeling and validating Simulink diagrams. *Formal Aspects of Computing*, 21(5):451–483, 2009.
- [8] W. Denman, M.H. Zaki, S. Tahar, and L. Rodrigues. Towards flight control verification using automated theorem proving. In *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 89–100. Springer Berlin Heidelberg, 2011.
- [9] C. Dixon, A. Winfield, and M. Fisher. Towards temporal verification of emergent behaviours in swarm robotic systems. In *Towards Autonomous Robotic Systems*, volume 6856 of *Lecture Notes in Computer Science*, pages 336–347, 2011.
- [10] J. Ezekiel, A. Lomuscio, L. Molnar, S. Veres, and M. Prebody. Verifying fault tolerance and self-diagnosability of an autonomous underwater vehicle. In *Proc. IJCAI*, pages 1659–1664, Barcelona, Spain, July 2011.
- [11] J.C. Filliâtre. One logic to use them all. In *Automated Deduction*, volume 7898 of *Lecture Notes in Computer Science*, pages 1–20, 2013.
- [12] M. Fisher, L. Dennis, and M. Webster. Verifying autonomous systems. *Communications of the ACM*, 56(9):84–93, September 2013.
- [13] H.D. Foster, A.C. Krolnik, and D.J. Lacey. *Assertion-Based Design*. Kluwer Academic Publishers, 2004.
- [14] D. Henrion, M. Ganet-Schoeller, and S. Bennani. Measures and LMI for space launcher robust control validation. In *Proc. Robust Control Design*, volume 7, pages 236–241, Denmark, 2012.

- [15] H. Herencia-Zapana, R. Jobredeaux, S. Owre, P.L. Garoche, E. Feron, G. Perez, and P. Ascariz. PVS linear algebra libraries for verification of control software algorithms in C/ACSL. In *NASA Formal Methods*, volume 7226 of *Lecture Notes in Computer Science*, pages 147–161, 2012.
- [16] Accellera Systems Initiative. *Accellera Standard OVL V2 Library Reference Manual*. Accellera Systems Initiative, 2013.
- [17] R. Jobredeaux, T. Wang, and E. Feron. Autocoding software with proofs I: Annotation translation. In *Proc. DASC*, pages 7C1–1–7C1–13, 2011.
- [18] M.E. Johnson. Model checking safety properties of servo-loop control systems. In *Proc. DSN*, pages 45–50, 2002.
- [19] S. Mitsch, K. Ghorbal, and A. Platzer. On provable safe obstacle avoidance for autonomous robotic ground vehicles. In *Proc. RSS*, Berlin, Germany, June 2013.
- [20] R. Muradone, D. Bresolin, L. Geretti, P. Fiorini, and T. Villa. Robotic surgery. *IEEE Robotics & Automation Magazine*, 18(3):24–32, September 2011.
- [21] A. Richards and J.P. How. Model predictive control of vehicle maneuvers with guaranteed completion time and robust feasibility. In *Proc. ACC*, volume 5, pages 4034–4040, June 2003.
- [22] A. Richards and J.P. How. Decentralized model predictive control of cooperating UAVs. In *Proc. CDC*, volume 4, pages 4286–4291, December 2004.
- [23] P. Roy and N. Shankar. SimCheck: a contract type system for Simulink. *Innovations in Systems and Software Engineering*, 7:73–83, 2011.
- [24] A.E. Rugina and J.C. Dalbin. Experiences with the GENE-AUTO code generator in the aerospace industry. In *Proc. ERTS*, 2010.
- [25] R. Simmons, C. Pecheur, and G. Srinivasan. Towards automatic verification of autonomous systems. In *Proc. IROS*, volume 2, pages 1410–1415, Takamatsu, Japan, November 2000.
- [26] G. Sirigineedi, A. Tsourdos, B.A. White, and R. Zbikowski. Kripke modelling and verification of temporal specifications of a multiple UAV system. *Annals of Mathematics and Artificial Intelligence*, 63(1):31–52, September 2011.
- [27] M. Staats and M.P.E. Heimdahl. Partial translation verification for untrusted code-generators. In *Formal Methods and Software Engineering*, volume 5256 of *Lecture Notes in Computer Science*, pages 226–237, 2008.
- [28] A. Toom, N. Izerrouken, T. Naks, M. Pantel, and O. Ssi Yan Kai. Towards reliable code generation with an open tool: Evolutions of the Gene-Auto toolset. In *Proc. ERTS*, Toulouse, France, May 2010.
- [29] W. Wang, P. Menon, D. Bates, S. Ciabuschi, N.M. Gomes Paulino, E. Di Sotito, A. Bidaux, A. Garus, A. Kron, S. Salehi, and S. Bennani. Verification and validation of autonomous rendezvous systems in the terminal phase. In *Proc. AIAA Guidance, Navigation, and Control Conference*, pages 1–11, Minneapolis, MN, USA, August 2012.
- [30] M. Webster, M. Fisher, N. Cameron, and M. Jump. Formal methods for the certification of autonomous unmanned aircraft systems. In *Formal Methods for the Certification of Autonomous Unmanned Aircraft Systems*, volume 6894 of *Lecture Notes in Computer Science*, pages 228–242, 2011.